XXVIII Congresso {virtual} de Iniciação Científica da Unicamp 2020

Estudo e Aplicação de Modelos de Previsão no Contexto de Séries de Vazões de Rios

Aluno: Daniel da Costa Nunes Resende Neto
Orientador: Prof. Levy Boccato
Faculdade de Engenharia Elétrica e de Computação (FEEC)

1 Introdução

Uma série temporal é uma sequência de medidas feitas ao longo do tempo sobre um fenômeno de interesse. Em várias áreas do conhecimento, tais como engenharia, economia, matemática aplicada e computação, diversas séries temporais são particularmente relevantes não só pela perspectiva de análise e interpretação de seus históricos passados, mas também pela possibilidade de estimar seus valores futuros. Este último desafio dá origem ao problema de predição de séries temporais [1].

Um cenário em que o problema de predição de séries temporais aparece com destacada importância está associado a séries de vazões de rios, no que diz respeito a um planejamento energético mais eficiente e seguro para uma determinada região. Isso se torna ainda mais relevante quando a principal fonte de geração de energia é constituída por usinas hidrelétricas, como é o caso do Brasil.

Neste contexto, foram estudados, a princípio, dois modelos lineares de predição clássicos da literatura: auto-regressivo (AR) e auto-regressivo de médias móveis (ARMA) [1]. No caso do modelo ARMA, exploramos uma meta-heurística denominada *clearing* [2] para o ajuste de seus parâmetros, visando minimizar o erro quadrático médio na etapa de treinamento.

Foram considerados também métodos não-lineares baseados em redes neurais [3]. São eles: a rede *perceptron* de múltiplas camadas (MLP, do inglês *multilayer perceptron*), como representante da classe de modelos *feedforward*; assim como três modelos recorrentes: as RNNs (do inglês, *recurrent neural networks*) simples, as LSTMs (do inglês, *long short-term memory*) e as GRUs (do inglês, *gated recurrent units*).

Todos os modelos foram aplicados às séries de vazões de Água Vermelha e Jirau (ver Seção 3.1), e, por fim, comparados a partir do Erro Quadrático Médio (EQM) gerado para os dados de teste de ambas as séries. A metodologia completa, os resultados obtidos e as conclusões finais do projeto encontram-se nas Seções 3.3 e 4 deste documento.

2 Modelos de Previsão

2.1 Modelos Lineares

2.1.1 AR

O modelo mais clássico de previsão é o auto-regressivo (AR, do inglês *auto-regressive*) [1, 4]. No AR , o valor da série no instante n, aqui denotado por $x(n)$, é determinado a partir de uma combinação linear dos valores passados até um instante $n - M - L + 1$, onde M determina a ordem do modelo e L o passo de previsão (quantos instantes de tempo à frente pretende-se estimar). Em termos matemáticos, a regra de evolução temporal do modelo AR é dada por:

$$x(n) = a_1 x(n-L) + \ldots + a_M x(n-M-L+1) + \eta(n) \quad (1)$$

onde $a_i, i = 1, \ldots, M$ são os coeficientes do modelo AR que ponderam as amostras passadas da série e $\eta(n)$ denota o erro instantâneo, cuja média é nula e cuja variância (σ_η^2) é constante. Este último termo constitui um ruído branco (em inglês, *white noise*) [1].

2.1.2 ARMA

A combinação das ideias presentes nos modelos AR e MA (do inglês, *moving-average*) [1] dá origem a um modelo linear mais geral, denominado ARMA (do inglês, *auto-regressive moving-average*) [1, 4]. Neste caso, o valor da série temporal no instante n depende linearmente de valores passados da própria série e também do ruído branco, como mostra a expressão abaixo:

$$x(n) = \sum_{k=0}^{M} a_k x(n-k-L+1) + \sum_{k=0}^{K} b_k \eta(n-k-J+1) \quad (2)$$

onde $b_0 = 1$, J e K são o passo e o número de atrasos do ruído branco, respectivamente, e as definições dadas para o modelo AR ainda valem.

Passaremos, agora, à apresentação dos fundamentos teóricos dos modelos não-lineares utilizados no trabalho.

2.2 Modelos Não-lineares

A definição de uma rede neural artificial passa pelas escolhas: (1) do modelo de neurônio, bem como (2) de sua função de ativação, (3) da arquitetura da rede, tipicamente organizada em camadas, e (4) dos valores mais apropriados para os hiperparâmetros, como o número de camadas e a quantidade de neurônios em cada camada. A seguir, apresentamos brevemente as estruturas estudadas neste projeto.

2.2.1 MLP

As MLPs são redes neurais formadas fundamentalmente por múltiplas camadas de neurônios artificiais [3], os quais recebem os atributos x_i contidos na instância $\mathbf{x}$ e os ponderam pelos pesos $w_i \subset \mathbf{w}$ do neurônio. Aplica-se, então, uma função de ativação ϕ sobre esta soma ponderada, acrescida de um termo de *bias* (b), de forma a gerar a saída:

$$h_{\mathbf{w}}(\mathbf{x}) = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right) \quad (3)$$

A função de ativação ϕ pode assumir várias formas, sendo as mais utilizadas as funções: logística (também chamada, em inglês, *sigmoid*), tangente hiperbólica (Tanh), ReLu e SeLu [3].

Uma MLP é composta, primeiramente, por uma camada ($l = 0$) de entrada. Os neurônios desta camada apenas passam para suas saídas o atributo com o qual foram alimentados. As próximas camadas, exceto a última, são denominadas camadas intermediárias (em inglês, *hidden layers*). Em cada camada, os estímulos de entrada são combinados linearmente, com base nos pesos de cada neurônio, e a função de ativação é aplicada sobre o resultado acumulado. Assim, cada uma das saídas dos neurônios de uma camada é propagada adiante para cada um dos neurônios da próxima camada, até que cheguem à última camada da rede.

Por fim, a camada de saída (em inglês, *output layer*) é quem gera as respostas da rede para uma instância. No treinamento da rede, discutido mais a diante, as instâncias são apresentadas à rede agrupadas em um *mini-batch*.

A equação (3) pode ser generalizada matricialmente (ver equações em (4)) para definir a matriz contendo os vetores de saídas de uma camada gerados a partir de uma sequência de vetores de entrada. Isto é feito separadamente para a camada de entrada (equação (4a)), uma camada intermediária (equação (4b)) e a camada de saída (equação (4c)). Essa representação possibilita o cálculo da matriz de saída de uma rede alimentada com um subconjunto de instâncias chamado *mini-batch*. Ela é utilizada no treinamento da rede (abordado em breve).

$$\mathbf{H}^0(\mathbf{X}) = \mathbf{X}, \tag{4a}$$

$$\mathbf{H}^l_{\mathbf{W^l},\mathbf{B^l}}(\mathbf{H}^{l-1}) = \phi(\mathbf{H}^{l-1}\mathbf{W}^l + \mathbf{B}^l), \tag{4b}$$

$$\mathbf{H}^L_{\mathbf{W^L},\mathbf{B^L}}(\mathbf{H}^{L-1}) = \phi(\mathbf{H}^{L-1}\mathbf{W}^L + \mathbf{B}^L) \tag{4c}$$

Nestas equações:

- A matriz $\mathbf{X} = [\mathbf{x}_1\ \mathbf{x}_2\ \dots\ \mathbf{x}_d]^\top$, com $\mathbf{X} \in \mathbb{R}^{d\times n}$, é o *mini-batch* recebido pela rede e contém d instâncias $\mathbf{x}_i^\top$ em suas linhas, sendo que cada instância é caracterizada por n atributos;
- Os vetores coluna $\mathbf{w}^l_j$ da matriz $\mathbf{W}^l = [\mathbf{w}^l_1\ \mathbf{w}^l_2\ \dots\ \mathbf{w}^l_{m(l)}]$, com $\mathbf{W}^l \in \mathbb{R}^{m(l-1)\times m(l)}$, contêm os conjuntos de pesos pertencentes a cada um dos $m(l)$ neurônios da camada l. Note que o número de pesos de um neurônio da camada l (a dimensão dos vetores $\mathbf{w}^l_j$) é igual ao número de neurônios da camada anterior $m(l-1)$, já que este é também seu número de saídas. Os valores assumidos por $m(l)$ quando $l = 1, ..., L$ são definidos na construção da rede e, como as saídas da camada de entrada são as próprias entradas, $m(0) = n$;
- A matriz $\mathbf{H}^l = [\mathbf{y}^l_1\ \mathbf{y}^l_2\ \dots\ \mathbf{y}^l_d]^\top$, com $\mathbf{H}^l \in \mathbb{R}^{d\times m(l)}$, contém os d vetores linha de saídas $\mathbf{y}^{l\top}_i$ da camada l. Cada um deles contém $m(l)$ saídas;
- Os vetores linha $\mathbf{b}^{l\top}_i$ da matriz $\mathbf{B}^l = [\mathbf{b}^l_1\ \mathbf{b}^l_2\ \dots\ \mathbf{b}^l_d]^\top$, com $\mathbf{B}^l \in \mathbb{R}^{d\times m(l)}$, são idênticos entre si e contêm os $m(l)$ valores de bias de cada neurônio da camada l;
- A função de ativação ϕ é aplicada a todos os elementos da matriz $(\mathbf{H}^{l-1}\mathbf{W}^l + \mathbf{B}^l)_{d\times m(l)}$, a qual contém em seu elemento (i, j) a soma dos elementos do i-ésimo (de um total de d) vetor de saída da camada anterior ponderados pelos pesos do j-ésimo neurônio da camada l, mais o termo de bias.

2.2.2 RNN

Uma RNN apresenta uma estrutura bastante similar à de uma MLP, mas com a presença de retroalimentações. Deste modo, a cada instante de tempo t (também chamado de *frame*), esta rede recebe o vetor de entradas $\mathbf{x}_{(t)}$ juntamente com o próprio vetor de saídas do instante de tempo anterior, $\mathbf{y}_{(t-1)}$.

Sendo assim, cada neurônio recorrente possui dois vetores de pesos: o primeiro, $\mathbf{w}_x$, pondera as entradas instantâneas $\mathbf{x}_{(t)}$; o segundo, $\mathbf{w}_y$, é responsável por ponderar as saídas passadas, $\mathbf{y}_{(t-1)}$. Considerando agora uma camada recorrente, podemos agrupar todos os pesos em duas matrizes, $\mathbf{W}_x$ e $\mathbf{W}_y$, de maneira que o vetor de saída da camada pode ser escrito como:

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{x}^\top_{(t)}\mathbf{W}_x + \mathbf{y}^\top_{(t-1)}\mathbf{W}_y + \mathbf{b}^\top\right)^\top \tag{5}$$

sendo $\mathbf{b}$ o vetor de bias.

De modo análogo ao que foi feito para a MLP, podemos agrupar várias instâncias em um único *mini-batch* $\mathbf{X}_{(t)}$, de modo que a matriz de saídas é dada por:

$$\begin{aligned}\mathbf{Y}_{(t)} &= \phi\left(\mathbf{X}_{(t)}\mathbf{W}_x + \mathbf{Y}_{(t-1)}\mathbf{W}_y + \mathbf{B}\right) \\ &= \phi\left(\begin{bmatrix}\mathbf{X}_{(t)} & \mathbf{Y}_{(t-1)}\end{bmatrix}\mathbf{W} + \mathbf{B}\right), \mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}\end{aligned} \tag{6}$$

onde a matriz $\mathbf{Y}_{(t)}$ contém em cada linha as saídas de uma instância do *mini-batch*; a matriz $\mathbf{X}_{(t)}$ contém em cada linha uma diferente instância de tamanho igual à quantidade de atributos; as matrizes $\mathbf{W}_x$ e $\mathbf{W}_y$ contêm em cada coluna os pesos que multiplicam os atributos e as saídas prévias de uma instância, respectivamente; a matriz $\mathbf{B}$ contém o mesmo vetor de bias de cada neurônio em cada uma de suas linhas; e as matrizes $\mathbf{W}_x$ e $\mathbf{W}_y$ podem ser concatenadas em uma única matriz $\mathbf{W}$, como mostrado.

Devido à relação de recorrência presente em (6), $\mathbf{Y}_{(t)}$ acaba sendo influenciado por todas as entradas desde $t = 0$, *i.e.*, $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \dots, \mathbf{X}_{(t)}$. Por isso, diz-se que a rede recorrente possui memória. A parte da rede que preserva estados ao longo do tempo é chamada de célula de memória (ou simplesmente célula).

A estrutura aqui apresentada (uma RNN simples) é uma célula muito básica, capaz apenas de aprender padrões curtos. As LSTMs e GRUs, apresentadas nas seções seguintes, buscam justamente a expansão desta memória.

2.2.3 LSTM

Vista de fora, uma célula LSTM pode ser utilizada analogamente a um neurônio recorrente simples. A diferença é que seu estado é dividido em dois vetores diferentes: o vetor de longo prazo $\mathbf{c}$ e o vetor de curto-prazo $\mathbf{h}$. A estrutura interna de uma célula LSTM está representada na Figura 1.

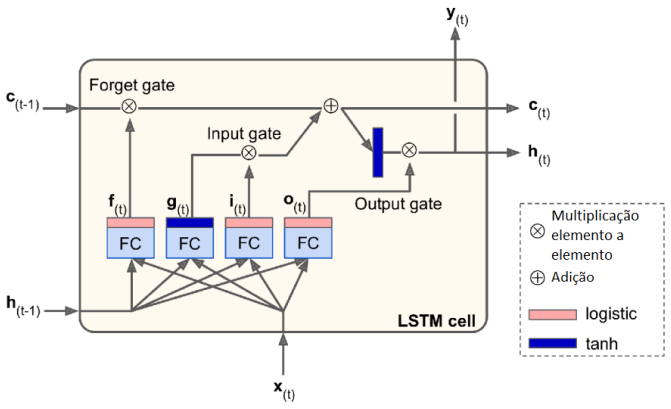

Figura 1: Estrutura interna de uma célula LSTM. Figura extraída de [3].

Já as equações em (7) mostram como computar a saída da célula, bem como seus estados, no instante t para um

mini-batch:

$$\mathbf{I}_{(t)} = \sigma(\mathbf{X}_{(t)}\mathbf{W}_{xi} + \mathbf{H}_{(t-1)}\mathbf{W}_{hi} + \mathbf{B}_i) \tag{7a}$$
$$\mathbf{F}_{(t)} = \sigma(\mathbf{X}_{(t)}\mathbf{W}_{xf} + \mathbf{H}_{(t-1)}\mathbf{W}_{hf} + \mathbf{B}_f) \tag{7b}$$
$$\mathbf{O}_{(t)} = \sigma(\mathbf{X}_{(t)}\mathbf{W}_{xo} + \mathbf{H}_{(t-1)}\mathbf{W}_{ho} + \mathbf{B}_o) \tag{7c}$$
$$\mathbf{G}_{(t)} = \tanh(\mathbf{X}_{(t)}\mathbf{W}_{xg} + \mathbf{H}_{(t-1)}\mathbf{W}_{hg} + \mathbf{B}_g) \tag{7d}$$
$$\mathbf{C}_{(t)} = \mathbf{F}_{(t)} \otimes \mathbf{C}_{(t-1)}^{\top} + \mathbf{I}_{(t)} \otimes \mathbf{G}_{(t)}^{\top} \tag{7e}$$
$$\mathbf{Y}_{(t)} = \mathbf{H}_{(t)} = \mathbf{O}_{(t)} \otimes \tanh(\mathbf{C}_{(t)}^{\top}) \tag{7f}$$

A primeira alteração que ocorre no estado a longo prazo do instante anterior $\mathbf{c}_{(t-1)}$ é a perda de parte de sua memória no que é chamado porta do esquecimento (do inglês, *forget gate*). Em seguida, é feita a incorporação de novas memórias pela porta de entrada (do inglês, *input gate*), formando o vetor $\mathbf{c}_{(t)}$ (equação (7e)). Uma cópia deste é filtrada pela porta de saída (do inglês, *output gate*) e resulta no estado de curto prazo $\mathbf{h}_{(t)}$ (equação (7f)), que também representa a saída da célula naquele instante.

Agora, olharemos para a entrada da LSTM. O vetor $\mathbf{g}_{(t)}$ é a saída da camada principal (equação (7d)). As partes mais relevantes de $\mathbf{g}_{(t)}$ são guardadas no vetor $\mathbf{c}_{(t-1)}$, e as menos relevantes descartadas.

Deste modo, o vetor $\mathbf{f}_{(t)}$ consegue controlar quais partes do vetor $\mathbf{c}_{(t-1)}$ devem ser apagadas (equação (7b)). O vetor $\mathbf{i}_{(t)}$ consegue controlar quais partes de $\mathbf{g}_{(t)}$ devem ser adicionadas ao vetor $\mathbf{c}_{(t-1)}$ (equação (7a)). Finalmente, o vetor $\mathbf{o}_{(t)}$ consegue controlar quais partes de $\mathbf{c}_{(t-1)}$ devem formar tanto a saída $\mathbf{y}_{(t)}$ quanto o estado de curto-prazo $\mathbf{h}_{(t)}$ (equação (7c)).

Resumidamente, uma célula LSTM é capaz de detectar uma entrada relevante (pela porta de entrada), guardá-la no vetor de estado a longo-prazo o quanto for necessário (pela porta do esquecimento) e extraí-la quando necessário (pela porta de saída). Isso traz à tona a capacidade elevada (em relação a uma célula recorrente simples) de reconhecimento e preservação de tendências presentes, no contexto deste projeto, em série temporais.

2.2.4 GRU

A célula GRU é uma versão simplificada da célula LSTM, e funciona de modo análogo. Sua estrutura interna está representada na Figura 2.

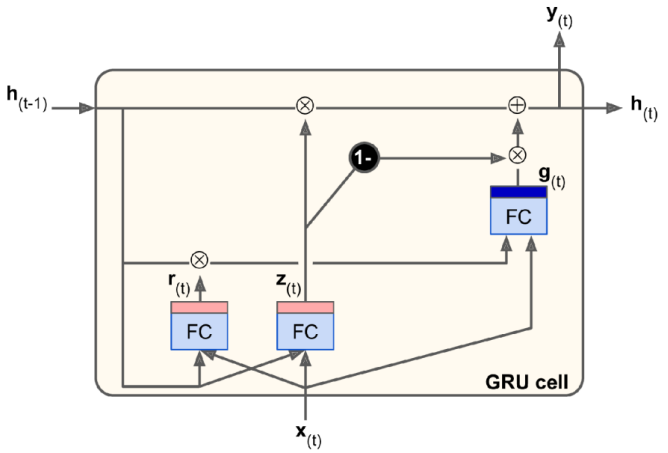

Figura 2: Estrutura interna de uma célula GRU. Figura extraída de [3].

Como pode ser notado, há agora somente um vetor de estado, $\mathbf{h}$. Também, um único vetor $\mathbf{z}_{(t)}$ controla tanto a porta de esquecimento (*forget gate*) quanto a porta de entrada (*input gate*). Em outras palavras, para que uma memória seja mantida, o conteúdo do local em questão deve ser primeiramente apagado.

Além disso, não existe mais uma porta de saída (*output gate*). Contudo, existe agora o vetor $\mathbf{r}_{(t)}$, o qual decide qual parte do vetor de estado prévio $\mathbf{h}_{(t-1)}$ será mostrado à camada principal (de saída $\mathbf{g}_{(t)}$). É importante notar que a capacidade de memória de uma LSTM ou uma GRU é, ainda assim, limitada, de modo que o aprendizado de padrões bastante afastados no tempo pode ser difícil.

2.2.5 Treinamento

Para que uma rede seja treinada, é necessário encontrar os valores de todos os pesos sinápticos, $\mathbf{W}^l, l = 1, \ldots, L$, que minimizem uma medida de erro (neste caso, o EQM) entre as saídas geradas pela rede e as saídas desejadas.

No caso da MLP, utiliza-se algoritmo de retropropagação do erro (em inglês, *error backpropagation*) [4]. Já para a RNN, aplica-se a retropropagação do erro através do tempo (do inglês, *backpropagation through time*).

O otimizador clássico utilizado na atualização dos pesos da rede é o gradiente descendente [3]. Contudo, a otimização por gradiente descendente sofre comumente com o desaparecimento dos gradientes (em inglês, *vanishing gradients problem*), ou ainda o contrário: o seu aumento desenfreado (em inglês, *exploding gradients problem*), que é mais comum no contexto de redes recorrentes.

Uma técnica desenvolvida para tentar evitar ambos os problemas supracitados é a normalização do *batch* (BN, do inglês *batch normalization*) [3]. Ela consiste na adição de uma operação dentro do modelo logo antes ou depois da função de ativação de cada camada intermediária.

Esta operação simplesmente centra todas as instâncias ao redor do zero e as normaliza. Para isso, o algoritmo deve estimar a média e o desvio padrão de cada atributo dentro do *mini-batch* (daí o nome) que alimenta a camada de *batch normalization*. Isso é feito da seguinte maneira:

$$\boldsymbol{\mu}_B = \frac{1}{d}\sum_{i=1}^{d}\mathbf{x}_i \tag{8a}$$
$$\boldsymbol{\sigma}_B^2 = \frac{1}{d}\sum_{i=1}^{d}(\mathbf{x}_i - \boldsymbol{\mu}_B)^2 \tag{8b}$$
$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}} \tag{8c}$$
$$\mathbf{z}_i = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}_i + \boldsymbol{\beta} \tag{8d}$$

onde $\boldsymbol{\mu}_B$ e $\boldsymbol{\sigma}_B$ são os vetores contendo, respectivamente, a média e o desvio padrão de cada atributo para o *mini-batch*; $\hat{\mathbf{x}}_i$ é a i-ésima instância centrada em zero e normalizada; $\boldsymbol{\gamma}$ é o vetor que pondera cada um dos atributos normalizados em $\hat{\mathbf{x}}_i$; $\otimes$ representa o produto elemento a elemento; o vetor $\boldsymbol{\beta}$ contém o *offset* relativo a cada atributo normalizado; ϵ é um número pequeno que impede a divisão por zero (tipicamente 10^{-5}); e $\mathbf{z}_i$ é a saída reescalada e deslocada da camada de *batch normalization*.

O uso de BN introduz dois novos parâmetros por camada: um que escala os atributos normalizados $\boldsymbol{\gamma}$ e outro que os desloca $\boldsymbol{\beta}$. Cabe ao modelo aprender a escala e o deslocamento ótimo de cada atributo para cada camada.

Outro problema característico do gradiente descendente é sua lentidão, a qual pode ser contornada com a utilização de outros otimizadores. O otimizador explorado neste trabalho é o Nadam (do inglês, *Nesterov adaptive moment estimation*).

O conjunto de equações que o descreve é dado por:

$$\mathbf{w}' = \mathbf{w} + \beta_3 \mathbf{m} \tag{9a}$$

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\mathbf{w}} E(\mathbf{w}') \tag{9b}$$

$$\mathbf{s} \leftarrow \beta_2 \mathbf{s} - (1 - \beta_2) \nabla_{\mathbf{w}} E(\mathbf{w}') \otimes \nabla_{\mathbf{w}} E(\mathbf{w}') \tag{9c}$$

$$\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1} \tag{9d}$$

$$\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2} \tag{9e}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \epsilon} \tag{9f}$$

A otimização por momento presente no Nadam incorpora a influência de gradientes passados, não apenas o local. Isto é feito subtraindo-se o gradiente local, desta vez, do vetor de momento $\mathbf{m}$ (ver equação (9b)). E este, por sua vez, é utilizado para atualizar o peso $\mathbf{w}$ (veja a equação (9f)).

O gradiente passa, então, a transmitir a ideia de aceleração, e não mais velocidade. Assim, o algoritmo acumula velocidade de convergência caso esteja caminhando para um ótimo local e escapa de platôs mais rapidamente.

Para que o algoritmo não avance velozmente em uma direção que não aponta para o ótimo local (como acontece no metódo clássico do gradiente descendente), introduz-se o vetor $\mathbf{s}$ (ver equação (9c)), o qual escala o vetor momento (ver equação (9f)).

Por fim, o Nadam também incorpora a ideia proposta pelo matemático Yurii Nesterov (daí o nome) de que o gradiente deve ser medido não sobre o ponto atual $\mathbf{w}$, mas sobre um ponto levemente à frente na direção do vetor momento, como faz a equação (9a). Esta ideia pode ser benéfica, uma vez que a direção deste aponta geralmente na direção correta (a direção do ponto ótimo), tornando a atualização dos pesos mais precisa.

Isto finaliza a exibição do embasamento teórico utilizado neste projeto. Nas próximas seções, apresentaremos características relevantes da séries de vazões, bem como a metodologia de treinamento aplicada. Concomitantemente, os resultados obtidos para ambos os modelos lineares e não-lineares na etapa de teste serão relatados, analisados e, ao final, comparados entre si.

3 Metodologia e Resultados

3.1 Descrição da Série e Divisão de Dados

As séries de vazões fluviais diárias (em m^3/s) utilizadas neste projeto foram as da usina de Água Vermelha e Jirau, considerando-se um período de 16 anos: de 1º de janeiro de 2000 a 31 de dezembro de 2015.

Os primeiros 5114 dias (1º de janeiro de 2000 a 31 de dezembro de 2013) constituíram os dados de treinamento e validação, os quais foram fornecidos aos modelos de previsão. Já os últimos 730 dias (1º de janeiro de 2014 a 31 de dezembro de 2015) foram usados na etapa de teste.

As vazões de cada uma das séries são significativamente diferentes, trazendo variedade às análises entre os métodos de previsão empregados.

3.2 Implementação

A mesma metodologia foi aplicada durante o projeto e avaliação dos modelos, de modo a tornar a comparação de seus respectivos desempenhos verossímil. A implementação destes foi dividida em duas etapas: validação e teste.

1. **Etapa de Validação**
   1.1. Escolhem-se números M de atrasos (valores passados da respectiva série) pertencentes a um intervalo I_M (definido a partir da função de autocorrelação parcial das séries), e treinam-se os modelos com validação cruzada (do inglês, *cross validation*) do tipo *holdout* ou *k-fold*. Para as redes neurais, especificamente, varia-se concomitantemente o número N de neurônios. Essas possuem todas uma única camada, porém diferem em relação à taxa de aprendizado, função de ativação, inicialização de dados e presença de BN. Já para o modelo ARMA, são considerados, também, números K de atrasos (valores passados dos erros cometidos) pertencentes a um intervalo I_K (definido a partir da função de autocorrelação das séries).
2. **Etapa de Teste**
   2.1. Ao final do treinamento, as melhores configurações de cada modelo são retreinadas, com ou sem validação, e utilizadas para prever múltiplas vezes todas as vazões presentes nos dados de teste (de 1º de janeiro de 2014 a 31 de dezembro de 2015) da série em questão. A média dos EQMs cometidos para os dados de teste é a medida do desempenho daquele modelo na previsão da respectiva série.

3.3 Comparação entre modelos

As melhores combinações de parâmetros de cada modelo e seus respectivos EQMs para os dados de teste são colocados na Tabela 1 em ordem crescente (para a série de Água Vermelha).

Tabela 1: Resultados Finais

Modelo	M	K	N	Água Vermelha $(m^3/s)^2$	M	K	N	Jirau $(m^3/s)^2$
RNN simples	5	-	20	1048 ± 110	4	-	5	414914 ± 11516
ARMA	5	1	-	1056,3	5	1	-	614060
GRU	10	-	100	1088 ± 98	9	-	100	502325 ± 40190
AR	29	-	-	1088,6	2	-	-	393130
MLP	8	-	10	1165 ± 79	4	-	5	478170 ± 6554
LSTM	8	-	150	1322 ± 100	22	-	100	456127 ± 20792

O primeiro ponto a ser destacado é que o desempenho das células recorrentes foi superior ao desempenho da MLP. Consideremos uma MLP com uma única camada de N_n neurônios. Esta rede contém, para instâncias de tamanho n, $n \times N_n$ pesos a serem ajustados. Já uma rede recorrente de camada única, e mesmo número de neurônios, necessita apenas ajustar N_n parâmetros (independentemente do tamanho da instância), pois estes são compartilhados entre

os n instantes de tempo. O número reduzido de pesos da rede recorrente simplifica seu aprendizado.

Quanto ao acesso das redes à instância, de fato, a MLP a observa por completo, enquanto a rede recorrente acessa somente um de seus elementos em cada instante de tempo. A informação introduzida por cada um desses elementos é passada para os instantes de tempo subsequentes apenas pelo estado da célula, dificultando a assimilação de padrões longos. Contudo, como as instâncias apresentam um tamanho $n \in I_M$ reduzido, a célula de memória consegue aprender com êxito os padrões fundamentais presentes na instância. Isto minimiza a possível vantagem possuída pela MLP, de acesso à instância completa, e configura outro motivo pelo qual o desempenho da RNN simples foi superior.

Por outro lado, tanto a LSTM quanto a GRU não ofereceram melhorias em relação à RNN simples. Apesar de serem capazes de manter uma informação dentro da rede por um período de tempo mais longo, provavelmente nos cenários aqui tratados tal memória de longo prazo não era estritamente necessária. Assim, houve um gasto de recursos com uma arquitetura mais flexível que provou ser infrutífero para a otimização destas células. Deste modo, ambas as propostas (LSTM e GRU) acabaram sendo superadas pela RNN simples.

Os modelos lineares, por sua vez, se saíram muito bem. O estudo feito em [5] mostra que modelos estatísticos (como o AR e o ARMA) podem superar métodos de aprendizado de máquina (ML, em inglês *Machine Learning*), cuja sofisticação e demanda computacional são maiores. Isso mostra que um alto desempenho de extrapolação é acompanhado de grande eficiência. Em outras palavras, uma capacidade superior de predição pode ser atingida a partir de uma menor complexidade computacional (CC, em inglês *Computational Complexity*). O inverso também é verdadeiro, já que abordagens mais complexas de extrapolação por métodos de ML exibem resultados piores, indicando que robustez teórica nem sempre é favorável.

O estudo [5] também mostra que modelos que melhor ajustam os parâmetros da rede durante o treinamento não necessariamente apresentam maior capacidade de extrapolação. O exemplo dado pelo artigo é a LSTM, a qual "comparada com redes neurais mais simples, como a RNN e a MLP, demonstrou melhor *fitting*, porém pior precisão de predição [para os dados de teste]". Comparando-se, neste projeto, o desempenho de validação da LSTM ao desempenho de validação da RNN simples, nota-se esse mesmo comportamento: os EQMs de validação da LSTM são, no geral, menores, porém a capacidade de extrapolação da RNN simples é superior no contexto de ambas as séries.

Em suma, as características das séries de Água Vermelha e Jirau foram bem representadas pelos modelos lineares AR e ARMA, uma vez que este obteve o segundo menor EQM final (1056,3 $(m^3/s)^2$) no contexto da série de Água Vermelha, e aquele obteve o menor EQM final (393130 $(m^3/s)^2$) no contexto da série de Jirau. Já a flexibilidade adicional dos modelos não-lineares acabou dificultando o aprendizado efetivo dos comportamentos das séries, com exceção da RNN simples, a qual apresentou um desempenho equiparável ao dos modelos lineares: o menor EQM final (1048 ± 110 $(m^3/s)^2$) no contexto da série de Água Vermelha e o segundo menor EQM final (414914 ± 11516 $(m^3/s)^2$) no contexto da série de Jirau.

4 Conclusões

Os avanços da inteligência artificial (AI, em inglês *Artificial Intelligence*) nos últimos anos permitiram a criação de veículos autônomos (AVs, em inglês *Autonomous Vehicles*), bem como de programas que dominam jogos (*e.g.*, xadrez, GO), que reconhecem e classificam expressões faciais, voz e escrita, que realizam traduções instantâneas, e que propõem diagnósticos médicos [6]. Em muitas destas aplicações, o desempenho atingido é tido como superior ao do próprio ser humano.

É possível que métodos de ML aplicados à previsão de séries temporais alcancem um desempenho igualmente sublime e superior ao dos modelos lineares. Contudo, a implementação desses métodos é mais difícil, pois as regras para a predição de uma série são desconhecidas e mutáveis, existem instabilidades estruturais nos dados considerados e há aplicações em que as próprias predições podem alterar o futuro, aumentando ainda mais a incerteza das estimativas. Logo, pesquisas adicionais que experimentem ideias inovadoras e criem ajustes aos modelos de previsão são necessárias para ultrapassar a capacidade de extrapolação dos métodos estatísticos.

Neste contexto, introduzir o aluno a variadas estruturas de redes neurais, algumas com apelo moderno, o põe em contato com o cenário atual do problema de previsão de séries por aprendizado de máquina, o qual se encontra ainda em desenvolvimento e possui ampla possibilidade de expansão, seja na área da estatística, econometria, matemática financeira, ou mesmo ambiental. Além disso, este projeto fornece os meios de compreensão básicos das diversas aplicações de redes neurais (por exemplo aquelas citadas anteriormente) e dá a ele a oportunidade de atuar, um dia, em uma dessas áreas.

Referências

[1] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*, 5th ed. Wiley, 2015.

[2] L. Boccato, "Aplicação de computação natural ao problema de estimação de direção de chegada," Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, 2010.

[3] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd ed. O'Reilly Media, Inc., 2019.

[4] S. Haykin, *Adaptive filter theory*, 5th ed. Prentice Hall, 2013.

[5] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "Statistical and machine learning forecasting methods: Concerns and ways forward," *PLOS ONE*, vol. 13, no. 3, pp. 1–26, 03 2018.

[6] H. Liang, B. Y. Tsui, and H. Ni, "Evaluation and accurate diagnoses of pediatric diseases using artificial intelligence," *Nat Med*, vol. 25, pp. 433–438, 2019.